Automated Test Generation from Vulnerability Signatures

Abdulbaki Aydin, Muath Alkhalaf, and Tevfik Bultan
Computer Science Department
University of California, Santa Barbara
Email: {baki,muath,bultan}@cs.ucsb.edu

***Abstract*—Web applications need to validate and sanitize user inputs in order to avoid attacks such as Cross Site Scripting (XSS) and SQL Injection. Writing string manipulation code for input validation and sanitization is an error-prone process leading to many vulnerabilities in real-world web applications. Automata-based static string analysis techniques can be used to automatically compute vulnerability signatures (represented as automata) that characterize all the inputs that can exploit a vulnerability. However, there are several factors that limit the applicability of static string analysis techniques in general: 1) undecidability of static string analysis requires the use of approximations leading to false positives, 2) static string analysis tools do not handle all string operations, 3) dynamic nature of the scripting languages makes static analysis difficult. In this paper, we show that vulnerability signatures computed for deliberately insecure web applications (developed for demonstrating different types of vulnerabilities) can be used to generate test cases for other applications. Given a vulnerability signature represented as an automaton, we present algorithms for test case generation based on state, transition, and path coverage. These automatically generated test cases can be used to test applications that are not analyzable statically, and to discover attack strings that demonstrate how the vulnerabilities can be exploited.**

I. Introduction

Correctness of input validation and sanitization operations is a crucial problem for web applications. One of the main forms of interaction between a user and a web application is through text fields. The text entered by the user is parsed by the web application and used as the input parameter for the action that is executed in response to the user's request. During action execution, user input can be passed as a parameter to security sensitive operations such as sending a query to the back-end database. If the input sent by the user inserts unintended commands to the generated database query (which is called SQL injection), then security of the application can be compromised resulting in unauthorized access to sensitive data or loss of data. In another attack scenario, called Cross Site Scripting (XSS), a user sends an input that stores malicious code in the database, that can later be used for attacking other users' machines. Even for input fields which are not entered as text fields (such as inputs that are entered using a drop box), a malicious user can change the input field and insert an attack by manipulating the http request that is generated by the browser.

In order to ensure the security of a web application, the user inputs that flow into security sensitive functions like databases queries must be correctly validated and sanitized. Unfortunately, web applications are notorious for security vulnerabilities such as SQL injection and XSS that are due to lack of input validation and sanitization, or errors in string manipulation operations used for input validation and sanitization.

In this paper, we present an automated testing framework that targets testing of input validation and sanitization operations in web applications for discovering vulnerabilities. Our framework combines automated testing techniques with static string analysis techniques for vulnerability analysis [1]. We use static string analysis to obtain an over-approximation of all the input strings that can be used to exploit a certain type of vulnerability. This set of strings is called a vulnerability signature, which could be an infinite set containing arbitrarily long strings.

For specification of different types of vulnerabilities we use attack patterns developed by security researchers. These are regular expressions that characterize the strings that would cause a vulnerability when sent to a security sensitive function. Given an attack pattern and a web application, we use automata-based string analysis techniques to generate an automaton that corresponds to the vulnerability signature for that application for the type of vulnerability characterized by the attack pattern. As input web applications, we use the deliberately insecure web applications that are developed by security researchers to demonstrate different types of programming practices that lead to vulnerabilities.

Using the vulnerability signature automata generated by analyzing the deliberately insecure web applications, we automatically generate test cases based on three coverage criteria: state, transition and path coverage. Each test case corresponds to a string such that, when that string is given as a text field input to a web application, it may exploit the vulnerability that is characterized by the given vulnerability signature. Our automated test generation algorithm tries to minimize the number of test cases while achieving the given coverage criteria.

In order to demonstrate the effectiveness of our approach we experimented on several real-world web applications. As we report later in the paper, the automatically generated test sets were very effective in identifying vulnerabilities in these applications.

The rest of the paper is organized as follows. In Section II we give an overview of our approach. In Section III we review

This research is supported in part by NSF grants CCF-0916112 and CNS-1116967. Muath Alkhalaf is funded in part by a fellowship from the King Saud University.

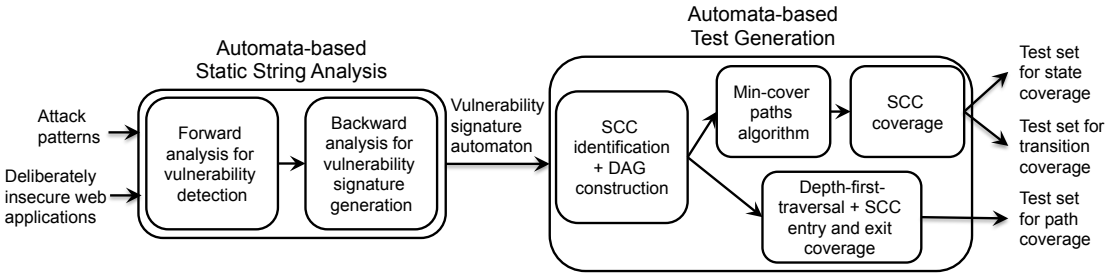

Figure 1. Automated Test Generation from Vulnerability Signatures

the vulnerability signature generation techniques we use. In Section V and VI we discuss the test generation algorithms we use. In Section VII we show the experimental results of our approach. In Section VIII we discuss the related work, and we conclude the paper in Section IX.

II. Motivation and Overview

The high-level flow of our automated testing framework for input validation and sanitization functions is shown in Figure 1. In this section we give an overview of different aspects of our approach, before explaining the technical details in the following sections.

A. Automata-based Static String Analysis

Our automated testing framework generates test cases from vulnerability signatures. A *vulnerability signature* is a characterization of all user inputs that can exploit a vulnerability. In our framework we use automata-based string analysis in which vulnerability signatures are represented as automata. Automata-based string analysis is a static program analysis technique. Given a set of input values represented as automata, it symbolically executes the program to compute the set of string values that can reach to each program point. Using a forward-analysis that propagates input values to sinks (i.e., security sensitive functions), it is possible to identify attack strings that can reach to a given sink. Then, a backward analysis that propagates the attack strings back to user input results in an automaton that corresponds to the vulnerability signature.

Automata-based static string analysis is challenging due to several reasons. Due to undecidability of string verification problem, string analysis techniques use conservative approximations that over-approximate the vulnerability signatures. Due to these approximations vulnerability signatures may contain strings that do not correspond to attacks, leading to false positives. Moreover, string analysis tools only model a subset of available string library functions, and when an unmodeled library function is encountered, the function has to be over-approximated to indicate that it can return all string values, which results in further loss of precision. Furthermore, forward and backward symbolic execution using automata can cause exponential blow-up in the size of the automata when complex string manipulation operations such as string-replace are used extensively. Finally, dynamic nature of scripting languages used in web application development makes static analysis very challenging and applicable to a restricted set of programs. Due to all these challenges it is not possible to have a push-button automata-based string analysis that works for all real-world applications.

In this paper we combine static vulnerability analysis techniques with automated test generation. The combined approach compensates for the weaknesses of the static vulnerability analysis techniques. In our approach static vulnerability analysis is applied to a small set of programs and the results from this analysis is used for testing other applications. Hence, programs with features that make static vulnerability analysis infeasible can still be checked using automated testing. Moreover, the approximations that are introduced by static vulnerability analysis that lead to false positives are eliminated during testing.

B. Generating Vulnerability Signatures from Deliberately Insecure Applications

Security researchers have developed applications that are deliberately insecure to demonstrate typical vulnerabilities. These applications are sometimes used to teach different pitfalls to avoid in developing secure applications, and sometimes they are used as benchmarks for evaluating different vulnerability analysis techniques. In our framework we use static string analysis techniques to analyze deliberately insecure applications and to compute a characterization of inputs that can exploit a given type of vulnerability.

In order to generate the vulnerability signature for an application, we need an attack pattern (specified as a regular expression) that characterizes a particular vulnerability. An attack pattern represents the set of attack strings that can exploit a particular vulnerability if they reach a sink (i.e., a security sensitive function). Attack patterns for different types of vulnerabilities are publicly available and can be used for vulnerability analysis.

Given an attack pattern and a deliberately insecure web application, we use automata-based static string analysis techniques to generate a vulnerability signature automaton that characterizes all the inputs for that application that can result in an exploit for the vulnerability characterized by the given attack pattern. I.e., the vulnerability signature automaton only accepts the strings that are in the vulnerability signature. In the next phase of our approach we automatically generate test cases from the vulnerability signature automaton.

C. Automated Test Generation from Vulnerability Signatures

Given a vulnerability signature automaton, any string accepted by the automaton can be used as a test case. Hence, any path from the start state of the vulnerability signature automaton to an accepting state characterizes a string which can be used as a test case. However, a vulnerability signature automaton typically accepts an infinite number of strings since, typically, there are an infinite ways one can exploit a vulnerability. In order to use vulnerability signature automata for testing, we need to somehow prune this infinite search space. Our overall goal is to minimize the number of test cases while making sure that we cover all possible ways of exploiting a vulnerability.

The mechanism that allows an automaton to represent an infinite number of strings is the loops in the automaton. So, in order to minimize the number of test cases, we have to minimize the way the loops are traversed. We do this by identifying all the strongly-connected components (SCCs) in an automaton and then collapsing them to construct a directed acyclic graph (DAG) that only contains the transitions of the automaton that are not part of an SCC and represents each SCC as a single node. Using this DAG structure, we do test generation for three coverage criteria: 1) *state coverage* where the goal is to cover all states of the automaton (including the ones in an SCC), 2) *transition coverage*, where the goal is to cover all transitions of the automaton (including the ones in an SCC), 3) *path coverage*, where the goal is to cover all the paths in the DAG that is constructed from the automaton, while also covering all possible ways to enter and exit from an SCC.

We implement the state and transition coverage using the min-cover paths algorithm that we execute on the DAG representation followed by a phase where we ensure the coverage of the states and transitions inside the SCC nodes. We implement the path coverage using depth-first-traversal, where, when an SCC node is encountered, we ensure that all entry and exit combinations are covered in the generated test cases.

D. A Sanitization Example

One of the well-known XSS attack strings is the following:
`<script>alert('XSS')</script>`
The script-tag indicates executable code and a malicious user might be trying to store a malicious script to be executed on another user's machine later on. Now, consider the example code in Figure 2 extracted from a deliberately insecure web application. This code is sanitizing the input provided by the user for the "name" field in line 7 by deleting all appearances of the string `<script>` (it deletes it by replacing each appearance of the string `<script>` with the empty string). Later on in the program, the variable `$html` is used as an input for a security sensitive function, so if the sanitization is not done properly this application would have a vulnerability.

We can try to check if the application is vulnerable by testing it with the above attack string. As expected the sanitization code will correctly remove the script-tag and sanitized input will be `alert('XSS')</script>`. So, this test input does not detect a vulnerability. However, this application has a vulnerability and the sanitization used in Figure 2 is incorrect.

```
1 <?php
2   if(!array_key_exists ("name", $_GET)
        || $_GET["name"] == NULL
        || $_GET["name"] == ""){
3     $isempty = true;
4   } else {
5     $html .= "<pre>";
6     $html .= "Hello ";
7     $html .=  str_replace( "<script>",
                             "",
                             $_GET["name"]);
8     $html .= "</pre>";
9   }
10 ?>
```

Figure 2. A Sanitization Example

One can generalize the attack strings for the XSS vulnerability as an attack pattern using the following regular expression:
`/.*<script.*>.*/`
When we run the automata-based string analysis on the example shown in Figure 2, we find out that the intersection of the set of strings that can reach the sink and the above attack pattern is not empty, i.e., there are some inputs that will cause a string containing the script-tag reach the sink. So, we generate the vulnerability signature for this application which results in an automaton that contains 59 states and 8530 transitions. Note that, this vulnerability signature automaton captures the fact that the string-replace operation in line 7 will delete all appearances of the string `<script>` from the input. The reason that there are thousands of transitions is due to the fact that there is a transition for each ASCII character from each state.

When we use our automated test generation technique to generate a test string from the vulnerability signature automaton, we obtain the following test input:
`<scrip<script>t>`
When we run the application with this input we discover an attack, i.e., the sink function receives an input that contains the string `<script>`. This is due to the fact that the incorrect sanitization function in Figure 2 deletes the substring `<script>` from the above test input and creates the attack string.

In our framework, we use the test strings generated from vulnerability signatures of deliberately insecure web applications to test other applications. If the applications we test contain sanitization errors similar to the errors in deliberately insecure web applications or if they do not use proper sanitization, then the generated test cases can discover their vulnerabilities without analyzing them statically. Note that the test inputs generated from vulnerability signatures can also be used for applications that are statically analyzable in order to eliminate false positives and construct exploits (i.e., to generate concrete inputs that demonstrate how a vulnerability can be exploited).

III. Vulnerability Signature Generation

We use an automata-based string analysis to generate the vulnerability signature from an application [2], [1]. This analysis takes as input a dependency graph for the input program. A dependency graph is a directed graph that specifies how the values of user inputs flow to the security sensitive functions

(sinks). The analysis consists of two phases. In the first phase, we perform a forward symbolic reachability analysis starting from nodes associated with input to compute all possible values that each node in the dependency graph can take. We use this information to collect vulnerable program points, as well as the reachable attack strings for those vulnerable program points. If the program is vulnerable, i.e., if there exists some vulnerable program points, we proceed to the second phase. In the second phase, we perform a backward symbolic reachability analysis from the vulnerable program points to compute all possible values of their predecessors that will result in attack strings at these vulnerable program points.

Figure 3 shows the algorithm used in our analysis. The algorithm takes three inputs: a dependency graph (denoted as G), a set of sink nodes (denoted as *Sink*), and an attack pattern (denoted as *Attk*). G is a directed dependency graph that specifies how the values of user inputs flow to the security sensitive functions. *Sink* denotes the nodes that are associated with security sensitive functions that might lead to vulnerabilities. *Attk* is a regular expression represented as an automaton that accepts the set of attack strings. At each node, the set of reachable string values is approximated as a regular language and represented symbolically as an automaton that accepts the language. To associate each node with its automaton, we create two automata vectors *POST* and *PRE*. The size of both is bounded by the number of nodes in G. *POST*$[n]$ is the automaton accepting all possible string values that can reach node n. *PRE*$[n]$ is the automaton accepting all possible string values that node n can take to exploit the vulnerability. Initially, all these automata accept nothing, i.e., their language is empty. *Vul* $\subseteq$ *Sink* is the set of vulnerable program points, and initially it is set to an empty set.

At line 4, we first compute *POST* by calling the forward analysis. At line 5, for each node $n \in$ *Sink*, we generate an automaton *tmp* by intersecting the attack pattern and the possible values of n. If the language of *tmp*, i.e., L(*tmp*), is not empty, we identify that n is a vulnerable program point and add it to *Vul* at line 8. In fact, *tmp* accepts the set of reachable attack strings at node n that can be used to exploit the vulnerability. Hence, we assign *tmp* to *PRE*$[n]$ at line 9. If *Vul* is not empty, we compute *PRE* by calling our backward analysis at line 13. Note that for $n \in$ *Vul*, *PRE*$[n]$ has been assigned. We report vulnerability signatures for each `input` node based on *PRE* at line 14-16. If *Vul* is an empty set, we report that the program is secure with respect to the attack pattern.

The forward symbolic reachability analysis is based on a standard work queue algorithm. We iteratively update the automata vector *POST* until a fixpoint is reached [2]. Backward analysis uses the results of the forward analysis. Particularly, it computes all possible values of each node n that can exploit the identified vulnerability. The challenge in both forward and backward analyses is computing pre and post-conditions of string manipulation functions such as concatenation, string-replace etc., where the inputs and outputs of the pre and post-condition operations are automata. We use the techniques described in [2] for pre and post-condition operations and the details of the symbolic automata-based forward and backward analyses can be found in [1].

The output of the vulnerability signature generation algorithm is a set of vulnerability signature automata. A vulnerability signature automaton is a tuple $V = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The alphabet Σ is the set of ASCII characters. Each transition $t \in \delta$ is a tuple $t = (q, c, q')$ where $q = source(t)$, $q' = target(t)$ and $c \in \Sigma$. The vulnerability signature automata are deterministic, i.e., there is a single transition for each source state and alphabet symbol.

```
1: procedure VulSigGeneration(G, Sink, Attk)
2:     Init(POST, PRE)
3:     Vul ← {}
4:     fwdAnalysis(G, POST)
5:     for all n ∈ Sink do
6:         tmp ← POST[n] ∩ Attk
7:         if L(tmp) ≠ ∅ then
8:             Vul ← Vul ∪ {n}
9:             PRE[n] ← tmp
10:        end if
11:    end for
12:    if Vul ≠ ∅ then
13:        bwdAnalysis(G, POST, PRE, Vul)
14:        for all n ∈ Input do
15:            ReportVulnerabilitySignature(PRE[n])
16:        end for
17:        return "Vulnerable"
18:    else
19:        return "Secure"
20:    end if
21: end procedure
```

Figure 3. Vulnerability Signature Generation

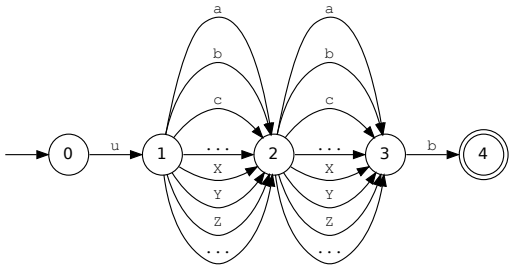

Figure 4. Large Number of Paths

IV. Converting Vulnerability Signature Automata to DAGs

Some features of the vulnerability signature automata make test generation difficult. One feature is that there are large number of transitions in δ where $source(t_0) = source(t_1) = source(t_2) = \ldots = source(t_n)$ and $target(t_0) = target(t_1) = target(t_2) = \ldots = target(t_n)$. Such transitions cause an exponential blow up in the number of accepting paths in the automaton, and this leads to a large search space for test generation. As an example consider state q_2 in Figure 4. For this relatively small automaton there are 128×128 accepting paths. Our solution to this problem is to collapse the transitions that have the same source and target states into one transition as shown in Figure 5. The label of the collapsed transition is a range of characters corresponding to each transition that it represents. During test generation we only pick one character from the range representing the all corresponding transitions.

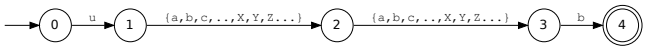

Figure 5. Collapsed Transitions

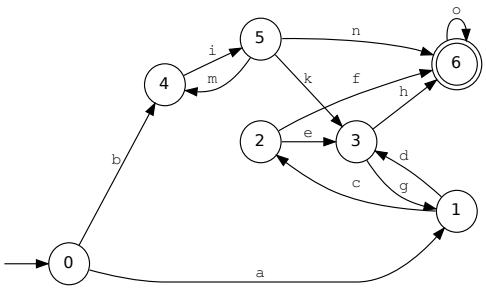

Figure 6. Cycles in Automata

This allows us to avoid exponential blow up in the number of accepting paths. For the rest of the paper we assume that all transitions with the same source and target states are collapsed.

Another feature of vulnerability signature automata is that they can contain cycles which results in an infinite number of accepting paths, i.e., an infinite search space for test generation. As an example, in Figure 6, states $\{q_1, q_2, q_3\}$ and $\{q_4, q_5\}$ form cycles. In order to bound the number of accepting paths and, therefore the search space for test generation, we extract a high level representation of the given vulnerability signature automaton by identifying its strongly connected components (SCC). The high level representation we obtain is a directed acyclic graph $DAG = (N, E)$ where N is the set of SCCs and E is the set of edges between SCCs. At the automaton level each edge $e \in E$ is a transition such that $source(e) \in scc_x, target(e) \in scc_y$ and $scc_x \neq scc_y$. We use Tarjan's strongly connected components algorithm to identify the cycles in the vulnerability signature automata [3]. The worst case time complexity of this algorithm is $O(|Q| + |\delta|)$ for a given vulnerability signature automaton $V = (Q, \Sigma, \delta, q_0, F)$. High-level DAG representation for the automaton in Figure 6 is shown in Figure 7. It consists of four strongly connected components $N = \{SCC_0, SCC_1, SCC_2, SCC_3\}$, and six edges among them $E = \{e_a, e_b, e_k, e_n, e_f, e_h\}$.

V. State and Transition Coverage for Vulnerability Signature Automata Using Min-Cover Paths Algorithm

In this section we discuss generating test cases from vulnerability signature automata based on state and transition coverage criteria. Given a vulnerability signature automaton $V = (Q, \Sigma, \delta, q_0, F)$, let $L(V)$ denote the set of strings accepted by V. Our aim is to find two sets of strings $S_{sc}, S_{tc} \subseteq L(V)$ that achieve state and transition coverage, respectively. The state and transition coverage definitions are as follows:

- For each state in $q \in Q$ there must be at least one string in S_{sc} such that the accepting path for that strings visits q.
- For each (collapsed) transition $t \in \delta$ there must be at least one string in S_{tc} such that the accepting path for that string includes t.

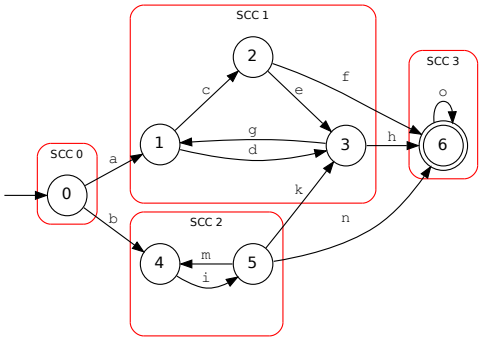

Figure 7. High Level DAG Representation

Finally, we want to generate the sets S_{sc} and S_{tc} in such a way that $|S_{sc}|$ and $|S_{tc}|$ are minimized.

The problem of finding minimum number of strings based on state and transition coverage criteria is very similar to a well-known graph problem called *minimum cover paths*. Given a directed acyclic graph, minimum cover paths is the least number of paths that visits each edge of the graph at least once. Minimum cover paths problem has been studied in different research areas and there are well known solutions to this problem [4], [5]. One known solution is to reduce minimum cover paths problem to the minimum flow problem [4], [6], [5]. We follow this basic approach with some modifications. We can divide the state and transition coverage algorithms into five main steps: 1) Initialization of DAG, 2) Converting DAG into a flow network, 3) Minimum flow algorithm, 4) Finding minimum covering paths, 5) Extending paths with SCC Coverage.

A. Initialization of DAG

Vulnerability signature automaton $V = (Q, \Sigma, \delta, q_0, F)$ has one start state q_0 and a set of final states F. In order to apply flow algorithms and minimum covering paths algorithm, one virtual final state q_v is added to Q, for each $q \in F$, a virtual transition $t_v = (q, \lambda, q')$ is added to the transition relation δ where λ is a new symbol added to the alphabet Σ. The modified automaton has one start state q_0 and one final state q_v. A DAG representation $DAG = (N, E)$ is constructed from the modified automaton as described in the previous section. We use $n_0 \in N$ to denote the start node of the DAG where $n_0 = SCC_0$ and $q_0 \in SCC_0$. Similarly, we use $n_v \in N$ to denote the as final node of the DAG such that $n_v = SCC_v$ and $q_v \in SCC_v$.

A vulnerability signature automaton always has a sink state that terminates non-accepting paths corresponding to non-accepting strings. As a result, corresponding DAG representation has a sink node that does not have any outgoing edges. We generate only the strings that are accepted by vulnerability signature automaton. To do so we remove the sink node and

```
1: procedure PREPROCESSRIGTHSC(node, queue)
2:     updated ← False
3:     for all edge ∈ outgoingEdges(node) do
4:         nextNode ← targetNode(edge)
5:         if flow(edge) = 0 then
6:             if #incomingEdges(nextNode) = 1 or
   #outgoingEdges(nextNode) = 1 then
7:                 flow(edge) ← 1
8:                 updated ← True
9:             else
10:                REMOVEFROMDAG(edge)
11:            end if
12:        end if
13:    end for
14:    if not updated or balanced(node) = 0 then
15:        return
16:    end if
17:    if updated and balanced(node) < 0 then
18:        queue.enqueue(node)
19:    else if updated and balanced(node) > 0 then
20:        DISTRIBUTEFLOWSEVENLY(node)
21:    end if
22:    for all edge ∈ outgoingEdges(node) do
23:        nextNode ← targetNode(edge)
24:        PREPROCESSRIGTH(nextNode, queue)
25:    end for
26: end procedure
```

Figure 8. Phase 1 for Pre-Processing of State Coverage

all incoming edges to the sink node from the DAG using a depth first traversal with a worst case complexity of $O(|E|)$.

B. Converting DAG into a Flow Network

A flow network is a DAG where each edge has a capacity and each edge receives a flow. Capacity for each edge $e \in E$ is a non-negative real value $c(e) \geq 0$. Flow is a function $f : E \rightarrow R$ that satisfies the following properties:

- For all $e \in E$, $f(e) \leq c(e)$.
- For all $e \in E, e' \in E$ where, $source(e) = target(e')$ and $target(e) = source(e')$, $f(e) = -f(e')$.
- For all $n \in N$, $\sum_{e \in incoming(n)} f(e) + \sum_{e' \in outgoing(n)} f(e') = 0$.

Min-cover paths algorithm does not require an upper bound for the capacity of an edge, and we assume that each edge has infinite capacity. We define a flow as the number of required visits to an edge in order to take each path from the start node to the final node. To apply the min-flow algorithm, we need an initial flow assignment for each edge in the DAG. We use a pre-processing algorithm [4] to assign an initial flow to each edge based on the number of input and output edges for each node. This is a two phase algorithm that consists of a depth first traversal starting from start node (Phase 1) followed by a reverse depth first traversal (Phase 2) if necessary. The first phase of the initialization for state coverage is shown in Figure 8.

The statement at line 6 checks for the edges that can be removed safely. For example edges labeled with $'f'$ and $'k'$ can be safely removed from Figure 7. The resulting high level DAG is shown in Figure 9. Depending on the order that for loop retrieves the edges at line 3, algorithm may remove different edges at different runs. However, this does not affect the state coverage.

We can define the flow function $flow(e)$ as number of visits for an edge $e \in E$. The $balanced()$ function compares the total input flow and total output flow for a node $n \in N$ based on flows for each incoming and outgoing edges. A positive balance means that the total input flow is larger than the total output flow. In that case line 20 distributes the input flows to the the output flows by updating the flow values of outgoing edges. For the case of a negative balance value, distribution is done in the reverse direction after Phase 1 finishes as described in [4]. Figure 9 also shows the initial flow values that are assigned to the example DAG. For the example shown in Figure 9, reverse pre-processing (Phase 2) is not necessary since in the first phase flows are already distributed correctly.

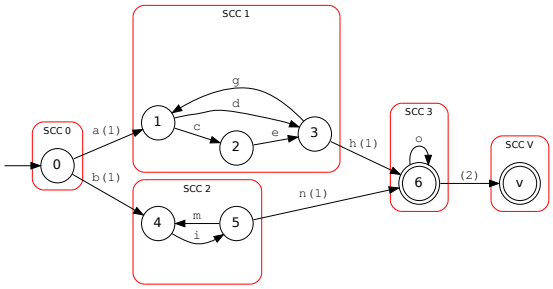

Figure 9. Initialized DAG for State Coverage

Phase 1 of the pre-processing algorithm for transition coverage is shown in Figure 10. The only modification compared to the algorithm shown in Figure 8 is inside the if block at line 5. The resulting flows for transition coverage are shown in Figure 11. Starting from the initial node, the algorithm first assigns a flow value of 1 to the edges $'a'$ and $'b'$. When it comes to SCC_2 during depth first traversal, it first assigns a flow of 1 to the edges $'k'$ and $'n'$. As a result balance value of SCC_2 becomes -1 and that SCC_2 is queued for reverse pre-processing. Similarly when algorithm first visits the SCC_1 using edges $'a'$ or $'k'$, balance value for SCC_1 becomes negative and SCC_1 is also queued for reverse pre-processing. However, when the algorithm visits SCC_1 for the second time, balance value becomes 0 and reverse pre-processing on SCC_1 does not have any effect.

C. Minimum Flow Algorithm

After we have initial flows calculated, Ford-Fulkerson algorithm is applied to the flow network with some modifications [7], [4]. Modified Ford-Fulkerson algorithm computes the minimum flows to visit each transition at least once. The algorithm finds paths from the start node to the final node and removes the maximum amount of flow from each path without reaching 0. Assume that our initialization phase calculated the flow for the path "bkh" in Figure 11 as "$b(4)k(3)h(3)$" instead of "$b(2)k(1)h(1)$". We can take away 2 flows from all the edges in the path "bkh". Time complexity of the algorithm for a DAG is $O(|p_{max}| \cdot (f_0 - f_{min}))$ where $|p_{max}|$ is the maximum length path from start node to final node, f_0 is initial flow set and f_{min} is the minimum flow [4].

```
1: procedure PREPROCESSRIGTHTC(node, queue)
2:     updated ← False
3:     for all edge ∈ outgoingEdges(node) do
4:         nextNode ← targetNode(edge)
5:         if flow(edge) = 0 then
6:             flow(edge) ← 1
7:             updated ← True
8:         end if
9:     end for
10:    if not updated or balanced(node) = 0 then
11:        return
12:    end if
13:    if updated and balanced(node) < 0 then
14:        queue.enqueue(node)
15:    else if updated and balanced(node) > 0 then
16:        DISTRIBUTEFLOWSEVENLY(node)
17:    end if
18:    for all edge ∈ outgoingEdges(node) do
19:        nextNode ← targetNode(edge)
20:        PREPROCESSRIGTH(nextNode, queue)
21:    end for
22: end procedure
```

Figure 10. Phase 1 for Pre-Processing of Transition Coverage

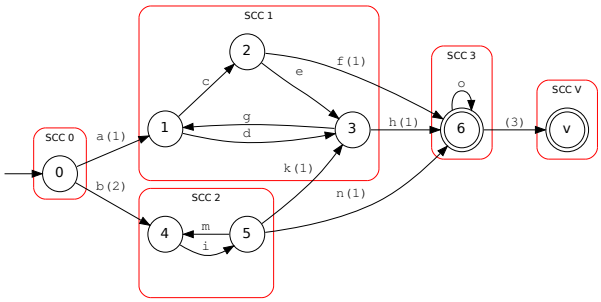

Figure 11. Initialized DAG for Transition Coverage

D. Finding Minimum Covering Paths

After running Minimum Flow Algorithm we can start looking for minimum covering paths. Minimum Covering Paths algorithm finds the edges that have $flow(e) > 0$ and forms a path that ends at the final node (i.e., the virtual node). Figure 12 shows the general loop and the recursive path finding function. For example, given the DAG shown in Figure 11, the minimum covering paths for transition coverage are computed as: "afe_v", "$bkhe_v$", and "bne_v" where e_v is the virtual edge.

Let N_k be the set of nodes that are k edges away from the start node. Let E_k be the set of edges between N_k and N_{k+1}. Let E_{max} be the edge set with maximum size among the sets $E_0, E_1, E_2, ...E_n$. Finally, let P_{max} be the maximum length path from start node to final node. Then, worst case time complexity for state and transition coverage is $O(|P_{max}| \times |E_{max}|)$ and the maximum size test set size for both coverage criteria is $O(|E_{max}|)$ which is equal to the number of minimum covering paths. For the DAGs that are extracted from the same vulnerability signature automaton let $|E_{max}|_{sc}$ denote the size of E_{max} for the DAG generated for state coverage and $|E_{max}|_{tc}$ denote the size of E_{max} for the DAG generated for transition coverage. Then, we have $|E_{max}|_{sc} \leq |E_{max}|_{tc}$. For the sets of test cases generated for state and transition coverage (S_{sc} and S_{tc}, respectively) we have $|S_{sc}| \leq |S_{tc}|$.

```
1: list minPaths ← NULL
2: loop
3:     path ← FINDMINPATH(node_start)
4:     if path = NULL then
5:         break
6:     else
7:         minPaths.add(path)
8:     end if
9: end loop
10: procedure FINDMINPATH(node)
11:     if node = node_final then
12:         path ← {}
13:         return path
14:     end if
15:     for all edge ∈ outgoingEdges(node) do
16:         if flow(edge) = 0 then
17:             continue
18:         end if
19:         DECREASEFLOWBYONE(edge)
20:         nextNode ← targetNode(edge)
21:         path = FINDMINPATH(nextNode)
22:         if path = NULL then
23:             continue
24:         end if
25:         path.add(edge)
26:         return path
27:     end for
28:     return NULL
29: end procedure
```

Figure 12. Minimum Covering Paths Algorithm

E. Extending Paths with SCC Coverage

Once we have the results for minimum covering paths we do a pass on each path and extend the SCC nodes $n \in N$ that represent cycles. We can define a strongly connected component as $SCC = (Q_{SCC}, \Sigma, \delta_{SCC})$ where $Q_{SCC} \subseteq Q$ and $\delta_{SCC} \subseteq \delta$. Assume there is a state $q_x \in Q_{SCC}$ and a transition $t \in \delta$. If $q_(x) = target(t)$ and $source(t) \notin Q_{SCC}$, we say state q_x is an entry point. Similarly, assume there is an edge $q_y \in Q_{SCC}$ and a transition $t \in \delta$. If $q_(x) = source(t)$ and $target(t) \notin Q_{SCC}$, we say state q_x is an exit point.

There are two different strategies for SCC coverage based on DAG coverage algorithm in progress. Strategy for the state coverage algorithm is the following: Starting from an entry point visit all states $q \in Q_{SCC}$ at least once and end up in an exit point. Similarly, for transition coverage starting from an entry point visit all transitions $t \in \delta_{SCC}$ at least once and end up at an exit point. If $|\delta_{SCC}|$ is greater than zero, then SCC must contain a cycle like SCC_1, SCC_2, and SCC_3 in Figure 7. To terminate the algorithm we keep a queue for unvisited states or unvisited transitions and use depth first search whenever necessary. Figure 13 shows the algorithm we use for state coverage. DFS function at line 7 starts a depth first search from the state given as its first argument and searches for the state given as its second argument without being trapped in a cycle. Once it finds the state given as its second argument, it returns a path that includes all the states it visited. Algorithm for visiting all transitions $t \in \delta_{SCC}$ is the same except we keep a queue for unvisited transitions instead of unvisited states. Both algorithms have a worst case complexity of $O(|\delta_{SCC}|^2)$ which depends on the overlapping cycles within a SCC. Worst case complexity of length of the returned path is also the same as the time complexity.

```
1: procedure VISITSTATES(SCC, q_entry, q_exit)
2:     list path ← NULL
3:     queue notVisited ← getAllStates(SCC)
4:     q ← q_entry
5:     notVisited.remove(q)
6:     while size(notVisited) ≠ 0 do
7:         visited ← DFS(q, notVisited.dequeue())
8:         notVisited.removeAll(visited)
9:         path.addAll(visited)
10:        q ← visited.last()
11:    end while
12:    if q ≠ q_exit then
13:        path.addAll(DFS(q, q_exit))
14:    end if
15:    return path
16: end procedure
```

Figure 13. SCC Coverage

Consider the example vulnerability signature automaton shown in Figure 9. Based on state coverage algorithm it can produce a path $.a.h.$ where each dot corresponds to a node in the DAG. Starting from the first dot which is actually SCC_0 we extend the path. SCC_0 returns an empty path and algorithm continues with next SCC in the path $a.h.$. SCC_1 returns ce for entry point q_1 and exit point q_3 and algorithm extends the path as $aceh.$. At the end the algorithm returns the extended path $aceh$.

VI. Path Coverage for for Vulnerability Signature Automata Using Depth First Traversal

A straight forward definition of path coverage would result in an infinite set of test cases due to loops in automata. So, given a vulnerability signature automaton V, we define $S_{pc} \subseteq L(V)$ as follows:

- For each path p in the DAG generated from V there must be a set of strings in S_{pc} such that the accepting paths for those strings must correspond to p (i.e. they must visit the same set of SCCs in the same order), and there must be an accepting path for each combination of entry and exit nodes for all the SCCs in the path p.

Path Coverage algorithm traverses DAG representation of vulnerability signature automata using a depth-first traversal (DFT). It does not have any initialization phase. It handles SCC entry-exit point coverage during path exploration. Assume current node in the DFT is n and n corresponds to a SCC. Again assume q_x is the entry point for the SCC corresponding to node n. Path coverage algorithm calculates paths for all possible combinations of q_x with all exit points using the SCC coverage algorithm we have for transition coverage. Then, it continues to explore paths in the high level DAG representation by following exit points in a DFT manner. By doing so, path coverage algorithm calculates all possible combinations of all entry and exit points of a SCC. The path coverage algorithm generates 5 paths for the example shown in Figure 11.

Based on definitions we have in previous section the time complexity for path coverage is $O(|E_{kmax}|^{P_{max}})$. Test size complexity is the same as the time complexity which is basically all paths from start node to final nodes. As a result we have the following test set size comparison for the three coverage criteria for the same vulnerability signature $|S_{sc}| \leq |S_{tc}| \leq |S_{pc}|$.

VII. Implementation and Experiments

In order to evaluate our automated testing framework, we used a deliberately insecure web application called Damn Vulnerable Web Application (DVWA) to generate vulnerability signatures. DVWA is listed in OWASP Broken Web Applications Project which lists deliberately insecure web applications. DVWA has several SQL injection, stored XSS and reflected XSS attacks with different security levels provided by the application. Security levels are no sanitization, custom sanitization, and incorrect use of built-in sanitization functions. We generated vulnerability signatures for each attack type considering different security levels. We used the Stranger static string analysis tool [8] to generate vulnerability signatures. We ran all the experiments on an Intel I5 machine with 2.5GHz X 4 processors and 32 GB of memory running Ubuntu 12.04.

Table I shows the properties of 5 vulnerability signatures generated from DVWA. We used the following well known attack patterns for vulnerability signature generation. Attack pattern `/.*<script.*>.*/` is used for vulnerability signatures `XSS 1`, `XSS 2`, and `XSS 3`. Attack pattern `/.* or 1 = 1 .*/` is used for vulnerability signature `SQLI 1` and attack pattern `/.*' or '1' = '1 .*/` is used for vulnerability signature `SQLI 2`. The sizes of the vulnerability signature automata depend on the complexity and number of string operations that application has on user inputs. We can see that vulnerability signatures `SQLI 1` and `XSS 1` are larger than the other three vulnerability signature automata. That is because the corresponding application code has more sanitization on user input. The application code that corresponds to vulnerability signature `SQLI 2` has no sanitization at all and the generated vulnerability signature is similar to the attack pattern. For each vulnerability signature, we can see that there is a big difference between the actual number of transitions that an automaton has and the corresponding number of collapsed transitions which allows us to reduce the sizes of the generated test sets. For a given vulnerability signature, the relation between the sizes of the test sets for different coverage criteria follows the ordering we expect where $|S_{sc}| \leq |S_{tc}| \leq |S_{pc}|$. For larger vulnerability signatures, path coverage algorithm produces a large number of strings as expected. For a given vulnerability signature, average length of the strings generated for state coverage is the smallest. Since the number of states are smaller than the number of transitions this is not surprising. The SCC coverage algorithm for state coverage produces strings with smaller lengths for most of the cases.

In order to evaluate the effectiveness of our automated test generation techniques we experimented on five open-source applications 1) PHP-Fusion v7.02.05 2 (content management system), 2) RuubikCMS v1.1.1 (website content management tool), 3) UL Forum v1.1.7 (forum application), 4) Snipe Gallery v3.1.5 (image management system), 5) PHP Server Monitor v2.0.1 (server management script). We implemented a web application driver to automatically execute the applications with the automatically generated test strings. We executed each application by assigning the automatically generated test strings to the selected vulnerable input fields. We enabled xdebug tool to record the server-side function call traces for each request that our web application driver sends. After each request, the web application driver extracts the sink function

Table I. VULNERABILITY SIGNATURE AUTOMATA

Vulnerability Signature	Automaton Size				Coverage	# of Strings	Avr. Len. for Generated Strings
	# of States	# of Transitions	# of Collapsed Transitions	# of SCCs			
SQLI 1	118	16327	574	19	State	8	39
					Transition	52	451
					Path Cov	321	437
SQLI 2	16	2649	58	3	State Cov	1	15
					Transition Cov	1	210
					Path Cov	1	210
XSS 1	100	13540	481	19	State Cov	8	31
					Transition Cov	44	312
					Path Cov	229	299
XSS 2	59	8530	237	3	State Cov	1	146
					Transition Cov	8	1,717
					Path Cov	8	1,628
XSS 3	11	1718	37	4	State Cov	1	10
					Transition Cov	2	73
					Path Cov	2	73

calls with values of parameters from the trace file. For the SQL injection attacks, each call to `mysql_query` function is treated as a sink function call. For the XSS attacks, each call to `mysql_query` function that executes `INSERT` or `UPDATE` statements is treated as sink function call. If the web application driver finds a sink function call, it checks the value of the query parameter of the sink function to confirm if it contains any type of attack.

Table II shows the effectiveness of the test sets generated using different coverage criteria on different applications. The sum of the third column and the fourth column shows the total number of test strings in a test set generated from all vulnerability signatures for a given coverage criteria. For example, there are a total of 19 test strings in the test set generated from all vulnerability signatures using the state coverage criteria. Third column shows the number of test strings that detected the vulnerability in the given application (stated in the first column), and the fourth column shows the number of test strings that missed the vulnerability. We can clearly say that path coverage and transition coverage have better detection rates than state coverage. The vulnerability detection rates for the applications *php_fusion* and *ruubik* are lower compared to other three applications for each coverage criteria. This is due to the fact that these applications have more string manipulation operations than the other three. For the fields selected from other three applications we observe the same detection rates. This is due to the fact that these applications all have the same type of vulnerability.

Table III shows the vulnerability detection rates of test sets generated using different coverage criteria for each vulnerability signature. It shows the distribution of the test sets in table II to different vulnerability signatures and different coverage criteria. Path coverage criteria has better detection rates for vulnerability signatures `XSS 1` and `SQLI 1` which are the larger vulnerability signature. For relatively small vulnerability signatures, path coverage and transition coverage detection rates are the same. Vulnerability signature `SQLI 2` has the worst detection rate. As we described previously in this section, that vulnerability signature is generated from a code that has no sanitization operations, which is not good enough for detecting attacks for applications that have some string operations. One interesting result is that state coverage for all `XSS` vulnerability signatures has a detection rate 0%. The application that we used to generate the vulnerability signatures concatenates HTML tags to the user inputs. Resulting vulnerability signature may include attack strings that has no closing tag $>$. State coverage generates only strings that do not have closing tags, but path and transition coverage criteria are able to handle that situation by visiting more transitions.

Table II. VULNERABILITY DETECTION PERFORMANCE PER APPLICATION

Application	Coverage Type	# Detected	# Missed	Detection Rate
ulforum	State	8	11	42%
	Transition	79	28	74%
	Path	477	84	85%
ruubik	State	4	15	21%
	Transition	28	79	26%
	Path	157	404	28%
php_fusion	State	2	17	11%
	Transition	42	65	39%
	Path	235	326	42%
snipe	State	8	11	42%
	Transition	79	28	74%
	Path	477	84	85%
phpservermon	State	8	11	42%
	Transition	79	28	74%
	Path	477	84	85%

Table III. VULNERABILITY DETECTION PERFORMANCE PER VULNERABILITY SIGNATURE

Application	Coverage Type	# Detected	# Missed	Detection Rate
SQLI 1	State	30	10	75%
	Transition	193	67	74%
	Path	1231	374	77%
SQLI 2	State	0	5	0%
	Transition	0	5	0%
	Path	0	5	0%
XSS 1	State	0	40	0%
	Transition	75	145	34%
	Path	553	592	48%
XSS 2	State	0	5	0%
	Transition	30	10	75%
	Path	30	10	75%
XSS 3	State	0	5	0%
	Transition	9	1	90%
	Path	9	1	90%

Overall, path coverage has better detection rates as expected. Transition coverage detection rates are very close to path coverage detection rates, and transition coverage generates smaller test sets. State coverage is not effective in generating attack strings for the vulnerability signatures we used.

VIII. Related Work

Static string analysis has been an active research area, with the goal of finding and eliminating security vulnerabilities caused by misuse of string manipulation operations [9], [10], [11], [12], [2], [13]. String analysis focuses on statically identifying all possible values of a string expression at a program point, and this knowledge can be leveraged to eliminate vulnerabilities such as SQL injection and XSS attacks. Due to undecidability of string analysis problem static string analysis approaches use conservative approximations such as widening [14], [15], [2], that can result in false positives. Moreover static modeling of all string manipulation functions is challenging and typically limits the applicability of static string analysis techniques. We are not aware of any prior work that combines static string analysis and vulnerability signatures with automated test generation.

In [16], [17], [18] dynamic symbolic execution has been used for automatic testing of a web application. First, string constraints are generated using symbolic execution. Then, these constraints are solved to generate vulnerable input strings. In [17], [18] authors use a bounded string constraint solver that bounds the length of the strings before solving the constraint. In [16] string constraints are represented using finite state transducers. Unlike dynamic symbolic execution, which is a white box testing approach, our approach is a black-box specification-based testing approach. Dynamic symbolic execution tries to increase execution path coverage while in our case we try to increase coverage of the vulnerability signature automaton that we use as a specification.

In [19] a black box SQLI/XSS web vulnerability scanner is developed utilizing manually written attack strings with no specific criteria. In XSS Analyzer [20], a black box testing approach is used where a very large database of attack strings is utilized to attack a web application. A learning algorithm is used to pick only a subset of this database. We use static analysis to automatically generate vulnerability signatures from which the attack strings are generated. Also, since we generate attack strings from an automaton, the original size of the attack string database could be infinite whereas in XSS analyzer the size of the attack string database is finite.

In [21] state machine based test generation using UML state charts is discussed. They define coverage criteria such as single UML transition coverage, full predicate coverage, transition-pair coverage, and complete sequence coverage. These coverage criteria are specific for UML diagrams. In [22] authors generate test cases from finite state machines that correspond to a software system specification. State machine based test generation has been used for different areas such as control systems, protocols, circuit design, data processing, navigation analyses.

Minimum cover paths algorithm has been studied for program testing [5] in order to generate minimum number of paths for certain features and to generate test data for those paths.

IX. Conclusion

We presented an automated testing framework for testing input validation and sanitization operations in web applications. In our framework the tests are generated from vulnerability signatures that are characterized as automata. Our experiments show that vulnerability signatures generated from deliberately insecure web applications can be used to generate effective tests for identifying vulnerabilities in other applications.

References

[1] F. Yu, M. Alkhalaf, and T. Bultan, "Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses," in *ASE*, 2009, pp. 605–609.

[2] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra, "Symbolic string verification: An automata-based approach," in *Proc. of SPIN*, 2008, pp. 306–324.

[3] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, 1972.

[4] M. Brandizi, N. Kurbatova, U. Sarkans, and P. Rocca-Serra, "graph2tab, a library to convert experimental workflow graphs into tabular formats," *Bioinformatics*, vol. 28, no. 12, pp. 1665–1667, 2012.

[5] S. C. Ntafos and S. L. Hakimi, "On path cover problems in digraphs and applications to program testing," *IEEE Trans. Software Eng.*, vol. 5, no. 5, pp. 520–529, 1979.

[6] E. Ciurea and L. Ciupal, "Sequential and parallel algorithms for minimum flows," *Journal of Applied Mathematics and Computing*, vol. 15, no. 1-2, pp. 53–75, 2004.

[7] L. Ford Jr and D. Fulkerson, "Maximal flow through a network," in *Classic papers in combinatorics*. Springer, 1987, pp. 243–248.

[8] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for php," in *TACAS*, 2010, pp. 154–157.

[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium, SAS '03*, ser. LNCS, vol. 2694. Springer-Verlag, June 2003, pp. 1–18.

[10] Y. Minamide, "Static approximation of dynamically generated web pages," in *Proceedings of the 14th International World Wide Web Conference*, 2005, pp. 432–441.

[11] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007, pp. 32–41.

[12] G.Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *ICSE*, 2008, pp. 171–180.

[13] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and Precise Sanitizer Analysis with Bek," in *Usenix Security Symposium*, 2011.

[14] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh, "A practical string analyzer by the widening approach," in *APLAS*, 2006, pp. 374–388.

[15] C. Bartzis and T. Bultan, "Widening arithmetic automata," in *Proceedings of the 16th International Conference on Computer Aided Verification*, 2004, pp. 321–333.

[16] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, 2008, pp. 249–260.

[17] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "Hampi: a solver for string constraints," in *ISSTA*, 2009, pp. 105–116.

[18] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *Proc. of the 31st IEEE Symposium on Security and Privacy (Oakland 2010)*, 2010.

[19] S. Kals, E. Kirda, C. Krügel, and N. Jovanovic, "Secubat: a web vulnerability scanner," in *WWW*, 2006, pp. 247–256.

[20] O. Tripp, O. Weisman, and L. Guy, "Finding your way in the testing jungle: a learning approach to web security testing," in *ISSTA*, 2013, pp. 347–357.

[21] A. J. Offutt and A. Abdurazik, "Generating tests from uml specifications," in *UML*, 1999, pp. 416–429.

[22] G. Friedman, A. Hartman, K. Nagin, and T. Shiran, "Projected state machine coverage for software testing," in *ISSTA*, 2002, pp. 134–143.